W3C Working Draft

Web Application Manifest

W3C Working Draft 13 August 2026

▼ **More details about this document**

This version:

https://www.w3.org/TR/2026/WD-appmanifest-20260813/

Latest published version:

https://www.w3.org/TR/appmanifest/

Latest editor's draft:

https://w3c.github.io/manifest/

History:

https://www.w3.org/standards/history/appmanifest/

Commit history

Editors:

Marcos Cáceres (Apple)

Daniel Murphy (Google Inc.)

Christian Liebel (Thinktecture AG)

Former editors:

Matt Giuca (Google Inc.) - Until 01 July 2024

Anssi Kostiainen (Intel Corporation) - Until 10 May 2024

Aaron Gustafson (Microsoft Corporation) - Until 10 May 2024

Mounir Lamouri (Google Inc.)

Rob Dolin (Microsoft Corporation)

Kenneth Rohde Christiansen (Intel Corporation) - Until 31 October 2025

Diego González (Microsoft Corporation) - Until 09 April 2026

Feedback:

GitHub w3c/manifest (pull requests, new issue, open issues)

Browser support:

caniuse.com

Abstract

This specification defines a JSON-based file format that provides developers with a centralized place to put metadata associated with a web application. This metadata includes, but is not limited to, the web application's name, links to icons, as well as the preferred URL to open when a user launches the web application. The manifest also allows developers to declare a default screen orientation for their web application, as well as providing the ability to set the display mode for the application (e.g., in fullscreen). Additionally, the manifest allows a developer to "scope" a web application to a URL. This restricts the URLs to which the manifest is applied and provides a means to "deep link" into a web application from other applications.

Using this metadata, user agents can provide developers with means to create user experiences that are more comparable to that of a native application.

Status of This Document

This section describes the status of this document at the time of its publication. A list of current W3C publications and the latest revision of this technical report can be found in the W3C standards and drafts index.

⚠ Warning

Implementors need to be aware that this specification is not stable. However, aspects of this specification are shipping in at least one browser (see links to implementation status at the top of this document). **Implementors who are not taking part in the discussions will find the specification changing out from under them in incompatible ways.** Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation phase should subscribe to the repository on GitHub and take part in the discussions.

This document was published by the Web Applications Working Group as a Working Draft using the Recommendation track.

Publication as a Working Draft does not imply endorsement by W3C and its Members.

This is a draft document and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to cite this document as other than a work in progress.

This document was produced by a group operating under the W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page

also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent that the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This document is governed by the 18 August 2025 W3C Process Document.

Table of Contents

§ 1. Web Application Manifest

An ***application manifest*** is a [*JSON*] document that contains startup parameters and application defaults for when a web application is launched.

A manifest has an associated ***manifest URL***, which is the [*URL*] from which the manifest was fetched.

A manifest can have any of the following members at its root, all of which are optional. The members can appear in any order.

- `background_color`
- `dir`
- `display`
- `icons`
- `id`
- `lang`
- `name`
- `orientation`
- `scope`
- `short_name`
- `shortcuts`
- `start_url`
- `theme_color`

> NOTE
>
> Although it is optional for any member to appear in a manifest, some user agents might require one or more to be present to take full advantage of the capabilities afforded by this specification.

§ 1.1 Examples

This section is non-normative.

This section shows how developers can make use of the various features of this specification.

§ 1.1.1 Typical structure

This section is non-normative.

The following shows a typical manifest.

EXAMPLE 1: Typical manifest

```
{
  "lang": "en",
  "dir": "ltr",
  "name": "Super Racer 3000",
  "short_name": "Racer3K",
  "icons": [{
    "src": "icon/lowres.webp",
    "sizes": "64x64",
    "type": "image/webp"
  }, {
    "src": "icon/lowres.png",
    "sizes": "64x64"
  }, {
    "src": "icon/hd_hi",
    "sizes": "128x128"
  }],
  "scope": "/",
  "id": "superracer",
  "start_url": "/start.html",
  "display": "fullscreen",
  "orientation": "landscape",
  "theme_color": "aliceblue",
  "background_color": "red"
}
```

§ 1.1.2 Using a `link` element to link to a manifest

This section is non-normative.

The example also shows how to use the link type "manifest" and how to use other `meta` and `link` elements to give the web application a fallback name and set of icons.

EXAMPLE 2: Linking to a manifest

```
<!doctype>
<html>
<title>Racer 3K</title>

<!-- Startup configuration -->
<link rel="manifest" href="manifest.webmanifest">

<!-- Fallback application metadata for legacy browsers -->
<meta name="application-name" content="Racer3K">
<link rel="icon" sizes="16x16 32x32 48x48" href="lo_def.ico">
<link rel="icon" sizes="512x512" href="hi_def.png">
```

NOTE: File extension: .webmanifest or .json?

The IANA registered file extension for the manifest is `.webmanifest`. Some web servers recognize this extension and transfer the file using the standardized application manifest media type (`application/manifest+json`). Developers can also choose a different extension (e.g. `.json`) or none at all (e.g. `/api/GetManifest`), but are encouraged to transfer the manifest using the `application/manifest+json` MIME type, although any JSON MIME type is ok.

§ 1.1.3 Declaring multiple icons

This section is non-normative.

This section illustrates how to declare multiple icons using the `icons` member to declare a set of icons for a web application. In the following example, the developer has made the following choices about the icons associated with the web application:

- The developer has included two icons at the same size, but in two different formats. One is explicitly marked as WebP through the `type` member. If the user agent doesn't support WebP, it can fall back to the second icon of the same size. The MIME type of this icon can then be either determined via a HTTP header, or can be sniffed by the user agent once the first few bytes of the icon are received.
- The developer specifies various sizes for the pixel-based icon formats (e.g., a PNG file). These sizes serve as hints for the user agent to determine a suitable icon to use in a particular context (e.g., on the home screen of a device). The developer has also included an ICO file (e.g., hd_hi.ico), which includes a range of raster icons individually tailored for particular display sizes. For example, it's often not suitable to simply downscale a 256x256 image to display in a 16x16 context, as the image will lose significant detail. Instead, an entirely different image specifically tailored for a 16x16 pixel context will often be used. Additionally, they added an SVG icon that can be resized dynamically to fit any icon size needed, but the tradeoff being that it can become unsuitable in some context (e.g., it becomes too small and blurry).

The list of icons is provided to the user agent, which will choose the most suitable icons for different contexts and placements.

EXAMPLE 3: Multiple icons

```
{
  "icons": [
    {
      "src": "icon/lowres.webp",
      "sizes": "48x48",
      "type": "image/webp"
    },{
      "src": "icon/lowres",
      "sizes": "48x48"
    },{
      "src": "icon/hd_hi.ico",
      "sizes": "72x72 96x96 128x128 256x256"
    },{
      "src": "icon/hd_hi.svg"
    }]
}
```

§ 1.1.4 Creating shortcuts

This section is non-normative.

In the following example, the developer has included two shortcuts. Assuming the manifest's URL is `https://example.com/manifest.webmanifest`:

- The first shortcut would be displayed with the text "Play Later". If the operating system supports icons for context menu items and it also supports SVG images for that purpose, the user agent would present `https://example.com/icons/play-later.svg` next to the text. When launched, the user agent would instantiate a new top-level traversable and navigate to `https://example.com/play-later`.
- The second shortcut would be displayed with the text "Subscriptions". When launched, the user agent would instantiate a new top-level traversable and navigate to `https://example.com/subscriptions?sort=desc`.

EXAMPLE 4: Adding shortcuts

```
{
  "shortcuts": [
    {
      "name": "Play Later",
      "description": "View the list of podcasts you saved for later",
      "url": "/play-later",
      "icons": [
        {
          "src": "/icons/play-later.svg",
          "type": "image/svg+xml"
        }
      ]
    },
    {
      "name": "Subscriptions",
      "description": "View the list of podcasts you listen to",
      "url": "/subscriptions?sort=desc"
    }
  ]
}
```

§ 1.1.5 Understanding "scope"

This section is non-normative.

The `scope` member tells the browser which documents are part of a web application, and which are not - and hence, to which set of web pages the manifest is "applied" when the user navigates around a web site.

For example, `{"scope": "/"}` means that the manifest applies to every document in an origin. On the other hand, `{"scope": "/racer/"}` means that only documents within the path "/racer/" are within scope: so "/racer/race1.html", "/racer/race2.html", etc. would all be within scope, but "/elsewhere/" and anything at the root "/" would be "out of scope" and the manifest wouldn't apply to documents in those paths. Only one scope path is supported. See 5. Navigation scope for the technical details.

Applying a manifest means that any members that affect presentation found in the manifest will come into effect, such as display "fullscreen", or applying a particular screen orientation. As long as the application is navigated to URLs that are within scope, the browser will continue to apply the manifest. However, navigating the web applications "out of scope" will cause the manifest to no longer be applied, and the browser will apply its own defaults. This will cause, for example, the application to no longer be displayed in fullscreen, and instead be displayed as a regular web page in a browser tab. It's left up to implementers to decide how to deal with web pages being navigated in and out of scope. See 1.17.5 Applying the manifest for the technical details.

Finally, as it's possible that a user can install a web application from any document within an origin, it's good practice to always declare a `scope` member in a manifest. If the `scope` member is missing from the manifest, then the path of the `start_url` member is used as a fallback. And if the `start_url` member is also missing, then the document URL from which the web application is installed gets used as the scope. To avoid any unexpected navigation behavior, authors should always include a `scope` member preferably set to `"/"`.

§ 1.2 `dir` member

The manifest's ***`dir`*** member specifies the ***default direction*** for the localizable members of the manifest. The `dir` member's value can be set to a text-direction.

The ***text-directions*** are the following, implying that the value of the localizable members is by default:

"*ltr*"
: Left-to-right text.

"*rtl*"
: Right-to-left text.

"*auto*" (default)
: Text direction is unknown. The user agent should use heuristics to estimate the display of the text, for example the first-strong algorithm as described in [*UAX9*].

The ***text-direction list*** is the list « "ltr", "rtl", "auto" ».

To ***process the `dir` member***, given ordered map *json* and ordered map *manifest*:

1. Set *manifest*["dir"] to "auto".
2. If *json*["dir"] doesn't exist or if *json*["dir"] is not a string, return.
3. Strip leading and trailing ASCII whitespace from *json*["dir"].
4. ASCII lowercase *json*["dir"].
5. If text-direction list doesn't contain *json*["dir"], return.
6. Set *manifest*["dir"] to *json*["dir"].

1.3 `lang` member

The manifest's ***`lang`*** member is a string in the form of a language tag that specifies the language for the values of the manifest's localizable members. If the `lang` member is not specified, the language is treated as unknown.

> NOTE
>
> Specifying the language improves the user experience by helping user agents select the most appropriate processing or resources, such as fonts, styling, hyphenation, or text-to-speech voices for accessibility.

A ***language tag*** is a string that matches the production of a well-formed `Language-Tag` defined in [*BCP47*].

> NOTE
>
> Language tags are case-insensitive. Examples of language tags include '`fr`' (French), '`en-AU`' (English as spoken in Australia), or '`zh-Hans-CN`' (Chinese as written in the Simplified Han script as spoken in China).

To ***process the `lang` member***, given ordered map *json* and ordered map *manifest*:

1. If *json*["lang"] doesn't exist or if *json*["lang"] is not a string, return.
2. Strip leading and trailing ASCII whitespace from *json*["lang"].
3. If calling IsStructurallyValidLanguageTag with *json*["lang"] returns `false`, return.
4. Set *manifest*["lang"] to the result of calling the CanonicalizeUnicodeLocaleId abstract operation with *json*["lang"].

§ 1.4 `name` member ▸ MDN

The manifest's ***`name`*** member is a string that represents the name of the web application as it is usually displayed to the user (e.g., amongst a list of other applications, or as a label for an icon).

The `name` member is a localizable member.

The `name` member serves as the accessible name of an installed web application.

> NOTE: Processing the `name` member
>
> When processing a manifest, the process a text member algorithm is used to process the `name` member.

§ 1.5 `short_name` member ▸ MDN

The manifest's ***`short_name`*** member is a string that represents a short version of the name of the web application. It is intended to be used where there is insufficient space to display the full name of the web application.

The `short_name` member is a localizable member.

> NOTE: Processing the `short_name` member
>
> When processing a manifest, the process a text member algorithm is used to process the `short_name` member.

§ 1.6 `scope` member ▸ MDN

The manifest's ***`scope`*** member is a string that represents the navigation scope of this web application's application context.

> NOTE: Default scope
>
> The "default scope" (when `scope` member is missing, empty, or failure) is the start URL, but with its filename, query, and fragment removed.

To ***process the `scope` member***, given ordered map *json* and ordered map *manifest*:

1. Set *manifest*["scope"] to the result of parsing "." with *manifest*["start_url"] as the base URL.
2. If *json*["scope"] is the empty string, then return.
3. Let *scope* be the result of parsing *json*["scope"] with *manifest URL* as the base URL.
4. If *scope* is failure, return.
5. Set *scope*'s query and fragment to null.
6. If *manifest*["start_url"] is not within scope of *scope*, return.
7. Otherwise, set *manifest*["scope"] to *scope*.

§ 1.7 `icons` member ▸ MDN

The manifest's ***`icons`*** member specifies images that serve as iconic representations of the web application in various contexts. For example, they can be used to represent the web application amongst a list of other applications, or to integrate the web application with an OS's task switcher and/or system preferences.

The `icons` member is a localizable member.

> NOTE
>
> When processing a manifest, the process image resources algorithm is used to process the `icons` member.

§ 1.8 `display` member

▸ MDN

The manifest's ***`display`*** member represents the developer's preferred display mode for the web application. Its value is a display mode.

To ***process the `display` member***, given ordered map *json* and ordered map *manifest*:

1. Set *manifest*["display"] to "browser".
2. If *json*["display"] doesn't exist or *json*["display"] is not a string, return.
3. Strip leading and trailing ASCII whitespace from *json*["display"].
4. ASCII lowercase *json*["display"].
5. If display modes list doesn't contain *json*["display"], return.
6. Set *manifest*["display"] to *json*["display"].

§ 1.9 `orientation` member

▸ MDN 🚫

The manifest's ***`orientation`*** member is a string that serves as the default screen orientation for all top-level traversables of the web application. The possible values are those of the `OrientationLockType` enum, which in this specification are referred to as the ***orientation values*** (i.e., "any", "natural", "landscape", "portrait", "portrait-primary", "portrait-secondary", "landscape-primary", or "landscape-secondary").

If the user agent supports the value of the `orientation` member as the default screen orientation, then that serves as the default screen orientation for the life of the web application (unless overridden by some other means at runtime). This means that the user agent *MUST* return the orientation to the

default screen orientation any time the orientation is unlocked [*SCREEN-ORIENTATION*] or the top-level traversable is navigated.

> NOTE
>
> Although the specification relies on the [*SCREEN-ORIENTATION*]'s `OrientationLockType`, it is *OPTIONAL* for a user agent to implement the [*SCREEN-ORIENTATION*] API. Supporting the [*SCREEN-ORIENTATION*] API is, of course, encouraged.

Certain UI/UX concerns and/or platform conventions will mean that some screen orientations cannot be used together. Which orientations and display modes cannot be used together is left to the discretion of implementers. For example, for some user agents, it might not make sense to change the default screen orientation of an application while in `browser` display mode.

> NOTE
>
> Once the web application is running, other means can change the orientation of a top-level traversable (such as via [*SCREEN-ORIENTATION*] API).

To ***process the `orientation` member***, given ordered map *json* and ordered map *manifest*:

1. If *json*["orientation"] doesn't exist or *json*["orientation"] is not a string, return.
2. Strip leading and trailing ASCII whitespace from *json*["orientation"].
3. ASCII lowercase *json*["orientation"].
4. If *json*["orientation"] doesn't contain any of the orientation values, return.
5. Set *manifest*["orientation"] to *json*["orientation"].

§ 1.10 `start_url` member

▸ MDN

The manifest's ***`start_url`*** member is a string that represents the ***start URL*** , which is URL that the developer would prefer the user agent load when the user launches the web application (e.g., when the user clicks on the icon of the web application from a device's application menu or homescreen).

The `start_url` member is purely advisory, and a user agent *MAY* ignore it or provide the end-user the choice not to make use of it. A user agent *MAY* also allow the end-user to modify the URL when, for instance, a bookmark for the web application is being created or any time thereafter.

To ***process the `start_url` member***, given ordered map *json*, ordered map *manifest*, URL *manifest URL*, and URL *document URL*:

1. Set *manifest*["start_url"] to *document URL*.
2. If *json*["start_url"] doesn't exist or *json*["start_url"] is not a string, return.
3. If the type of *json*["start_url"] is not string, or if *json*["start_url"] is the empty string, return.
4. Let *start URL* be the result of parsing *json*["start_url"], using *manifest URL* as the base URL.
5. If *start URL* is failure, return.
6. If *start URL* is not same origin as *document URL*, return.
7. Otherwise, set *manifest*["start_url"] to *start URL*.

> EXAMPLE 5
>
> For example, if the value of `start_url` is `../start_point.html`, and the manifest's URL is `https://example.com/resources/manifest.webmanifest`, then the result of parsing would be `https://example.com/start_point.html`.

§ 1.10.1 Privacy consideration: `start_url` tracking

It's conceivable that the `start_url` could be crafted to indicate that the application was launched from outside the browser (e.g., `"start_url": "index.html?launcher=homescreen"`). This can be useful for analytics and possibly other customizations. However, it is also conceivable that developers could encode strings into the start_url that uniquely identify the user (e.g., a server-assigned identifier, such as `"?user=123"`, `"/user/123/"`, or `"https://user123.foo.bar"`). This is fingerprinting/privacy sensitive information that the user might not be aware of.

> NOTE: Don't add identifiers to start URLs
>
> It is bad practice for a developer to use the start URL to include information that uniquely identifies a user, as it would represent a fingerprint that is not cleared when the user clears site data. However, nothing in this specification can practically prevent developers from doing this.

Given the above, it is *RECOMMENDED* that, upon installation, or any time thereafter, a user agent allows the user to inspect and, if necessary, modify the start URL of an application.

A user agent *MAY* offer other protections against this form of fingerprinting. For example, if a user clears data from an origin, the user agent *MAY* offer to uninstall applications that are within scope of that origin, thus removing the potential fingerprint from the application's start URL.

§ 1.11 `id` member

▸ MDN

The manifest's ***`id`*** member is a string that represents the ***identity*** for the application. The identity takes the form of a URL, which is same origin as the start URL.

The identity is used by user agents to uniquely identify the application universally. When the user agent sees a manifest with an identity that does not correspond to an already-installed application, it *SHOULD* treat that manifest as a description of a distinct application, even if it is served from the same URL as that of another application. When the user agent sees a manifest where *manifest*["id"] is equal (with exclude fragments OPTIONALLY set to true) to the identity of an already-installed application, it *SHOULD* be used as a signal that this manifest is a replacement for the already-installed application's manifest, and not a distinct application, even if it is served from a different URL than the one seen previously.

> NOTE: Excluding fragments is best practice
> Since the processing algorithm removes the fragment from the `id` member, it is not strictly necessary to exclude fragments when checking for a matching application. However, since old versions of this spec (and, possibly, old user agents) did not remove the fragment from the URL at parse time, and relied only on excluding fragments during comparisons, historical app data could contain fragments in the `id`. Therefore, it is best practice for user agents to exclude fragments even when comparing two URLs that ought not to have fragments.

> NOTE
>
> The identity can be used by a service that collects lists of web applications to uniquely identify applications.

> NOTE
>
> The identity is processed like a URL but it doesn't point to a resource that can be navigated to, so it's not required to be within scope.

To ***process the `id` member***, given ordered map *json*, ordered map *manifest*:

1. Set *manifest*["id"] to *manifest*["start_url"].
2. If the type of *json*["id"] is not string, return.
3. If *json*["id"] is the empty string, return.
4. Let *base origin* be *manifest*["start_url"]'s origin.
5. Let *id* be the result of parsing *json*["id"] with *base origin* as the base URL.
6. If *id* is failure, return.
7. If *id* is not same origin as *manifest*["start_url"], return.
8. Set *id*'s fragment to null.
9. Set *manifest*["id"] to *id*.

EXAMPLE 6: Resulting ids

The table below shows some example `id`s resulting from the process the `id` member steps.

***json*["id"]**	***manifest*["start_url"]**	***manifest*["id"]**
undefined	"https://example.com/my-app/start"	"https://example.com/my-app/sta
undefined	"https://example.com/my-app/#here"	"https://example.com/my-app/"
""	"https://example.com/my-app/start"	"https://example.com/my-app/sta
"/"	"https://example.com/my-app/start"	"https://example.com/"
"foo"	"https://example.com/my-app/start"	"https://example.com/foo"
"foo?x=y"	"https://example.com/my-app/start"	"https://example.com/foo?x=y"
"foo#heading"	"https://example.com/my-app/start"	"https://example.com/foo"
"./foo"	"https://example.com/my-app/start"	"https://example.com/foo"
"https://example.com/foo"	"https://example.com/my-app/start"	"https://example.com/foo"
"https://anothersite.com/foo"	"https://example.com/my-app/start"	"https://example.com/my-app/sta
"😀"	"https://example.com/my-app/start"	"https://example.com/%F0%9F%

Since `id` is resolved against `start_url`'s origin, providing "../foo", "foo", "/foo", "./foo" all resolves to the same identifier. As such, best practice is to use a leading "/" to be explicit that the id

is a root-relative URL path. Also, standard encoding/decoding rules apply to the id processing algorithm, as per the *URL Standard*.

§ 1.12 `theme_color` member

▸ MDN

The manifest's ***`theme_color`*** member is a themeable member that serves as the ***default theme color*** for an application context. What constitutes a ***theme color*** is defined in [*HTML*].

If the user agent honors the value of the `theme_color` member as the default theme color, then that color serves as the theme color for all top-level traversables to which the manifest is applied. However, the user agent *MAY* override the default theme color if a document whose URL is within scope of the application context's manifest includes a `meta` element whose `name` attribute is "`theme-color`". However, the user agent *SHOULD NOT* override the default theme color via a `meta` element whose `name` attribute is "theme-color" for documents' URL are not within scope, since the application has no control over these documents.

The user agent *MAY* ignore the theme color's alpha component based on the context. For example, in most environments, the theme color cannot be transparent.

Implementors *MAY* override the value defined by the `theme_color` member to support prefers-color-scheme.

NOTE

When processing a manifest, the process a color member algorithm is used to process the `theme_color` member.

§ 1.13 `background_color` member

▸ MDN

The manifest's ***`background_color`*** member is a themeable member that describes the expected background color of the web application. It repeats what is already available in the application stylesheet but can be used by the user agent to draw the background color of a web application for which the manifest is known before the files are actually available, whether they are fetched from the network or retrieved from disk.

The `background_color` member is only meant to improve the user experience while a web application is loading and *MUST NOT* be used by the user agent as the background color when the web application's stylesheet is available.

Implementors *MAY* override the value defined by the `background_color` member to support prefers-color-scheme.

> NOTE
>
> When processing a manifest, the process a color member algorithm is used to process `background_color` member.

§ 1.14 `shortcuts` member

▸ MDN

The manifest's ***`shortcuts`*** member is an list of shortcut items that provide access to key tasks within a web application.

> NOTE
>
> Shortcuts could, for instance, be used to link directly to a user's timeline within a social media application or to their recent orders in an e-commerce context.
>
> Developers are encouraged to order their shortcuts by priority, with the most critical shortcuts appearing first in the list.

How shortcuts are presented, and how many of them are shown to the user, is at the discretion of the user agent and/or operating system.

To ***process the `shortcuts` member***, given ordered map *json*, ordered map *manifest*, and URL *manifest URL*:

1. Let *processedShortcuts* be a new list.
2. Set *manifest*["shortcuts"] to *processedShortcuts*.
3. If *json*["shortcuts"] doesn't exist or *json*["shortcuts"] is not a list, return.
4. For each *entry* of *json*["shortcuts"]:

1. Let *shortcut* be process a shortcut with *entry*, *manifest URL*, *manifest*["scope"], and *manifest*["dir"].
2. If *shortcut* is failure, continue.
3. Append *shortcut* to *processedShortcuts*.

A user agent *SHOULD* expose shortcuts via interactions that are consistent with exposure of an application icon's context menu in the host operating system (e.g., right click, long press). A user agent *SHOULD* render the shortcuts in the same order as they are provided in the manifest. A user agent *SHOULD* represent the shortcuts in a manner consistent with exposure of an application icon's context menu in the host operating system. A user agent *MAY* truncate the list of shortcuts presented in order to remain consistent with the conventions or limitations of the host operating system.

§ 1.15 `*_localized` members

A ***localizable member*** is a manifest member that can be localized. Each localizable member of the manifest has a corresponding ***`*_localized`*** member, where * represents the member name.

> EXAMPLE 7
>
> For example, the `name` member is a localizable member, and `name_localized` is its corresponding `*_localized` member.

A ***language map*** is an ordered map whose key is a language tag and whose value is a localized value. The ***localized value*** is content localized in the language given by the key.

EXAMPLE 8: Localizing the application name

```
{
  "lang": "en-US",
  "dir": "ltr",
  "name": "Color Picker",
  "name_localized": {
    "de":    "Farbwähler",
    "en":    {"value": "Color Picker"},
    "en-GB": {"value": "Colour Picker", "dir": "ltr"},
    "fr":    {"value": "Sélecteur de Couleur", "lang": "fr-CA", "dir": "
    "ar":    {"value": "منتقي الألوان", "dir": "rtl"}
  }
}
```

The value assigned to the localizable member is the ***default representation***. `*_localized` members contain a language map that defines localized values for the given localizable member in the application. The user agent *SHOULD* use the user's localization settings to select the localized value whose language tag key best matches the user's preference. When no such localized value is available, the default representation is used.

§ 1.15.1 Localizing text values

A ***localized text object*** is an ordered map with the following properties:

`value`
: The localized string.

***`dir`* (optional)**
: The text-direction.

***`lang`* (optional)**
: A language tag.

For localizable members that accept strings, the `*_localized` member's language map accepts either a string or a localized text object as the localized value.

When a string is used, or when the `dir` member of the localized text object is missing, the default direction (`dir` member of the manifest) is applied.

NOTE

The `dir` member of the localized text object needs to be present if the direction of a localized string differs from the default direction set in the manifest (`dir` member). Right-to-left text will require specific direction settings if the manifest's default direction is left-to-right, and vice versa.

When a string is used, or when the `lang` member of the localized text object is missing, the language tag of the language map key is applied.

NOTE

To support multilingual content and ensure optimal display and accessibility, it is possible to specify a different language for a localized text object. This is needed for situations where a term or text needs to be presented in a language different from the user's set locale.

EXAMPLE 9

For example, brand names might need to be pronounced in a language different from the user's locale:

```
{
  "lang": "fr",
  "name": "Superbes biscuits",
  "name_localized": {
    "de-DE": {"value": "Super Cookies", "lang": "en"}
  }
}
```

To ***process a `*_localized` text member***, given ordered map *json*, ordered map *map*, string *member*, and text-direction *defaultDirection*:

1. If *member* does not exist in *json*, return.
2. Let *languageMap* be *json*[*member*].
3. If *languageMap* is not an ordered map, return.
4. Let *languageTags* be the keys of *languageMap*.
5. Set *map*[*member*] to a new ordered map.
6. For each *languageTag* of *languageTags*, run process a localized text object, passing *languageMap*[*languageTag*], *languageTag*, *map*, *member*, and *defaultDirection*.

To ***process a localized text object***, given string or ordered map *localizedValue*, string *defaultLanguageTag*, ordered map *map*, string *member*, and text-direction *defaultDirection*:

1. Let *normalizedValue* be an ordered map.
2. If *localizedValue* is a string, strip leading and trailing ASCII whitespace from *localizedValue* and set *normalizedValue*["value"] to *localizedValue*.
3. If *localizedValue* is an ordered map:
    1. If "value" exists in *localizedValue* and *localizedValue*["value"] is a string, strip leading and trailing ASCII whitespace from *localizedValue*["value"] and set *normalizedValue*["value"] to *localizedValue*["value"].
    2. If "lang" exists in *localizedValue* and *localizedValue*["lang"] is a string, strip leading and trailing ASCII whitespace from *localizedValue*["lang"] and set *normalizedValue*["lang"] to *localizedValue*["lang"].
    3. If "dir" exists in *localizedValue* and *localizedValue*["dir"] is a string:
        1. Strip leading and trailing ASCII whitespace from *localizedValue*["dir"].
        2. If text-direction list contains *localizedValue*["dir"], set *normalizedValue*["dir"] to *localizedValue*["dir"].
4. If "value" does not exist in *normalizedValue*, return.
5. If "lang" does not exist in *normalizedValue*, set *normalizedValue*["lang"] to *defaultLanguageTag*.
6. If "dir" does not exist in *normalizedValue*, set *normalizedValue*["dir"] to *defaultDirection*.
7. If calling IsStructurallyValidLanguageTag with *normalizedValue*["lang"] or calling IsStructurallyValidLanguageTag with *defaultLanguageTag* returns `false`, return.
8. Set *map*[*member*][*defaultLanguageTag*] to *normalizedValue*.

> NOTE
>
> The process a localized text object algorithm takes both a string or a localized text object for the localized value parameter, but the processed result will be normalized into a localized text object with the `value`, `lang`, and `dir` members set.

§ 1.15.2 Localizing image resources

For localizable members that accept a list of image resources, the `*_localized` member's language map accepts a list of image resources as the localized value.

EXAMPLE 10: Localizing the application icon

```
{
  "lang": "en-US",
  "icons": [
    { "src": "icon/lowres.png", "sizes": "64x64" },
    { "src": "icon/hires.png", "sizes": "256x256" }
  ],
  "icons_localized": {
    "fr": [
      { "src": "icon/lowres_fr.png", "sizes": "64x64" },
      { "src": "icon/hires_fr.png", "sizes": "256x256" }
    ]
  }
}
```

To ***process a `*_localized` image resource member***, given ordered map *json*, ordered map *map*, string *member*, and URL *manifest URL*:

1. If *member* does not exist in *json*, return.
2. Let *languageMap* be *json*[*member*].
3. If *languageMap* is not an ordered map, return.
4. Let *languageTags* be the keys of *languageMap*.
5. Set *map*[*member*] to a new ordered map.
6. For each *languageTag* of *languageTags*:
    1. If calling IsStructurallyValidLanguageTag with *languageTag* returns `false`, continue.
    2. Run process image resources, passing *languageMap*[*languageTag*] as the list of image resources, *map*[*member*], *manifest URL*, and *languageTag* as the member.

§ 1.16 `color_scheme_dark` member

The manifest's ***`color_scheme_dark`*** member is an ordered map whose keys are themeable members and whose values are the overriding color values for those members when the operating system uses a dark color theme.

A ***themeable member*** is one of the following manifest members:

- `theme_color`
- `background_color`

When applying a manifest and the operating system uses a dark color theme, for each themeable member *member* that exists in `color_scheme_dark`, the user agent *SHOULD* use the value of `color_scheme_dark`[*member*] in place of *member*'s value, unless the user's preferences, such as accessibility settings, take precedence.

> EXAMPLE 11: Example manifest with a dark theme
>
> ```
> {
>   "background_color": "#fff",
>   "theme_color": "red",
>   "color_scheme_dark": {
>     "background_color": "#000",
>     "theme_color": "hotpink"
>   }
> }
> ```

§ 1.16.1 Theming themeable members

To ***process the `color_scheme_dark` member***, given ordered map *json*, ordered map *manifest*, and URL *manifest URL*:

1. If *json*["color_scheme_dark"] does not exist, return.
2. Let *colorScheme* be *json*["color_scheme_dark"].
3. If *colorScheme* is not an ordered map, return.
4. Let *processedColorScheme* be a new ordered map.

5. Set *manifest*["color_scheme_dark"] to *processedColorScheme*.
6. For each *member* of « "theme_color", "background_color" »:
    1. Process a color member passing *colorScheme*, *processedColorScheme*, *member*.

§ 1.17 Manifest life-cycle

This section defines algorithms for processing a manifest, and applying a manifest.

A user agent *MUST* support the link type "manifest" and the associated steps for how to fetch and process the linked resource.

§ 1.17.1 Processing the manifest

When instructed to ***ignore***, the user agent *MUST* act as if whatever manifest, member, or value caused the condition is absent.

The following algorithm provides an ***processing extension-point***: other specifications that add new members to the manifest are encouraged to hook themselves into this specification at this point in the algorithm. They *SHOULD NOT* modify the existing values already in the *manifest* object.

> NOTE
>
> The processing extension-point is meant to help avoid issues related to monkey patching.

The steps for ***processing a manifest*** are given by the following algorithm. The algorithm takes a URL *document URL*, a URL *manifest URL*, a byte sequence *bodyBytes*, and *client* (an environment settings object or null).

1. Let *json* be the result of parse JSON bytes to an Infra value passing *bodyBytes*.
2. If the *json* is a parsing exception, or *json* is not an ordered map:
    1. Set *json* to an empty ordered map.
3. Let *manifest* be an empty ordered map.
4. Process the `dir` member passing *json* and *manifest*.
5. Process the `lang` member passing *json* and *manifest*.

6. Process a text member passing *json*, *manifest*, and "name".
7. Process a `*_localized` text member passing *json*, *manifest*, "name_localized", and *manifest*["dir"].
8. Process a text member passing *json*, *manifest*, and "short_name".
9. Process a `*_localized` text member passing *json*, *manifest*, "short_name_localized", and *manifest*["dir"].
10. Process the `start_url` member passing *json*, *manifest*, *manifest URL*, and *document URL*.
11. Process the `id` member passing *json* and *manifest*.
12. If the document's processed manifest is not null, and document's processed manifest's id is not equal to *manifest*["id"], return.
13. Process the `scope` member passing *json*, *manifest*, and *manifest URL*.
14. Process a color member passing *json*, *manifest*, and "theme_color".
15. Process a color member passing *json*, *manifest*, and "background_color".
16. Process the `display` member passing *json* and *manifest*.
17. Process image resources passing *json*["icons"], *manifest*, *manifest URL*, and "icons".
18. Process a `*_localized` image resource member passing *json*, *manifest*, "icons_localized", and *manifest URL*.
19. Process the `color_scheme_dark` member passing *json*, *manifest*, and *manifest URL*.
20. Process the `orientation` member passing *json*, *manifest*.
21. Process the `shortcuts` member passing *json*, *manifest*, and *manifest URL*.
22. Processing extension-point: process any proprietary and/or other supported members at this point in the algorithm.
23. Set *manifest*'s ***client*** to *client*.
24. Let document's ***processed manifest*** be *manifest*.

§ 1.17.2 Processing color members

> NOTE: Supported colors
>
> Only sRGB colors, and colors the user agent can convert to sRGB without any outside knowledge (e.g., `"AliceBlue"`), are supported. For example, `lab(…)` or `color(display-p3, …)` can be converted to sRGB without outside knowledge, but `color(--custom-profile, …)` would require finding a matching "@color-profile" rule which cannot be specified in the manifest.

To ***process a color member***, using ordered map *json*, ordered map *map*, and string *member*:

1. If *json*[member] doesn't exist or *json*[member] is not a string, return.
2. Strip leading and trailing ASCII whitespace from *json*[*member*].
3. Let *color* be the result of parsing the value of *json*[member] as a CSS color.
4. If *color* is failure, return.
5. If *color* can be converted to sRGB using solely information the user agent inherently knows, then convert *color* to sRGB.
6. If *color* is not sRGB color, return.
7. Set *map*[member] to *color*.

§ 1.17.3 Processing text members

To ***process a text member***, given ordered map *json*, ordered map *map*, and string *member*:

1. If *json*[*member*] doesn't exists or *json*[*member*] is not a string, return.
2. Strip leading and trailing ASCII whitespace from *json*[*member*].
3. Set *map*[member] to the value of *json*[*member*].

§ 1.17.4 Processing the manifest without a document

The processing a manifest steps are invoked by [*HTML*]'s processing steps for the `link` element, in which case *client* is the document's relevant settings object. The steps *MAY* also be invoked by the user agent to process a manifest without an associated document, in which case *client* is null.

In this case, to match the guarantees made by the corresponding steps in [*HTML*], the user agent *SHOULD* ensure that at least at some point in the past:

- there was at least one document that is same origin as *document URL* that has a `link` element *linkElement* with a `rel` of `manifest` and a `href` that resolves to *manifest URL*, and that
- if *manifest URL* is not same origin as *document URL*, the *bodyBytes* data was fetched with a CORS Request whose `Origin` is *document URL*'s origin, and whose credentials mode is set to the CORS settings attribute credentials mode for *linkElement*'s `crossorigin` attribute.

> NOTE: The rationale for these checks
>
> This provision allows user agents to perform app installations using a manifest without having to first load a document that links to the manifest (e.g. when installing an app onto the user's device via a sync service). But the above recommendations ask the user agent to verify that, when an application scope is on origin A and the manifest is on a different origin B, there is a bidirectional relationship between the two origins.
>
> The first check ensures that at least some page on origin A links to the manifest on origin B (otherwise, it would be possible for a manifest on origin B to control the metadata for an unaffiliated app on origin A).
>
> The second check ensures that origin B allows (via the CORS protocol) its manifest to be applied by origin A, and also that the manifest is fetched with or without credentials, as agreed by both origins, as it would if going directly through the document.

§ 1.17.5 Applying the manifest

A processed manifest is ***applied*** to a top-level traversable, meaning that the members of the manifest are affecting the presentation and/or behavior of the top-level traversable. Whenever a top-level

traversable is created, the user agent *MAY* apply a manifest to it before navigation begins.

A user agent *MAY* also apply a manifest to an existing top-level traversable. If the active document's URL is within scope of the manifest, the existing top-level traversable becomes an application context. If the URL is not within scope, the resulting behavior is implementation-defined.

> NOTE
>
> Some user agents might apply a manifest to the existing top-level traversable from which installation was initiated. If the active document's URL is not within scope of the manifest's navigation scope, one possible behavior is that the application context initially exists at an out-of-scope URL.

> NOTE
> Whether to apply a manifest, and which manifest to apply, is at the discretion of the user agent, based on the user's actions. For example, if the user launched an application from the system menu or from a shortcut, the user agent might create a new top-level traversable with that application's manifest applied, but it might not do so if the user simply clicked a bookmark to a URL within the application's navigation scope.

A top-level traversable that has a manifest applied to it is referred to as an ***application context***.

A processed manifest can also be ***unapplied*** from a top-level traversable. When this happens, the top-level traversable ceases to be an application context.

> NOTE
>
> A user agent might unapply a manifest, for example, when the user removes the web application from their device while a document associated with the application context is still open. The top-level traversable's subsequent behavior (e.g., whether it closes, reverts to a regular browser tab, etc.) is implementation-defined.

If an application context is created as a result of the user agent being asked to navigate to a deep link, the user agent *MUST* immediately navigate to the deep link with *historyHandling* set to "`replace`". Otherwise, when the application context is created, the user agent *MUST* immediately navigate to the start URL with *historyHandling* set to "`replace`".

NOTE

The start URL is not necessarily the value of the `start_url` member: the user or user agent could have changed it when the application was installed.

§ 1.17.6 Updating the manifest

As specified for `manifest` link relation, the manifest is fetched and processed on every page load. When processing a manifest is successful, user agents *MAY* apply the updated manifest to any current and future application contexts associated with the application.

The user agent *SHOULD* consider a manifest image resource updated if the `src` member has changed. If the `src` has not changed, the user agent *MAY* download the image and check for visual differences in some cases.

NOTE: Icon metadata changes

The way a manifest image resource is parsed for updates is similar to the `Cache-Control:immutable` behavior outlined in [*RFC8246*].

For the purpose of updating, the following members are ***security-sensitive members***, as they are presented during installation and on launch surfaces:

1. `short_name` and localized representations in `short_name_localized`,
2. `icons` and localized representations in `icons_localized`,
3. `name` and localized representations in `name_localized`.

All other members of the manifest are considered as ***non-security-sensitive members***.

A ***security-sensitive update*** is an update to a security-sensitive member. Respectively, a ***non-security-sensitive update*** is an update to a non-security-sensitive member.

When considering an updated security-sensitive member of type manifest image resource (e.g. `icons`), the user agent *MAY* consider it a non-security-sensitive update if the user agent finds the image not significantly visually different.

The user agent *SHOULD* apply all non-security-sensitive updates immediately.

The user agent *SHOULD* present all security-sensitive updates to the user and require express permission before applying the changes.

> NOTE: Example user options displayed when presented with the update
>
> The user may be presented with the following options when shown security-sensitive updates:
>
> 1. Accept the update
> 2. Uninstall the web app
> 3. Ignore the update

Security-sensitive members *SHOULD* be displayed in a bidirectionally isolated way as described in [*UTS55*], regardless of their direction.

If a user changes localization settings, the user agent *MAY* automatically adjust the security-sensitive members visible on launch surfaces to their localized representations specified in the `*_localized` members. These changes *SHOULD* be presented to users the next time they open the web application.

§ 2. Manifest image resources

Each ***manifest image resource*** is an image resource. The context in which a manifest image resource is presented is determined by the semantics of the associated manifest member (e.g., an `icons` member is generally used to represent the application icon).

A manifest image resource differs from a image resource in that it can have an additional `purpose` member.

User agents *MAY* modify the images associated with a manifest image resource to better match the platform's visual style before displaying it to the user, for example by rounding the corners or painting it in a specific color. It is recommended that developers prepare their image resources for such scenarios to avoid losing important information through, e.g., change of color or clipped corners.

To ***fetch a manifest image resource*** given a manifest image resource *image* and an application manifest *manifest*, return either the result of fetching an image resource or null:

1. If *manifest*'s client is null, return null.

> NOTE
> When a manifest is processed without an associated document, *manifest*'s client is null and no image fetch is performed. This ensures that enforcement of CSP and service worker fetch interception, which depend on the document's relevant settings object, remain in effect.

2. Let *request* be a new request.
3. Set *request*'s URL to *image*'s `src`.
4. Set *request*'s destination to "`image`".
5. Set *request*'s client to *manifest*'s client.
6. Return the result of fetching an image resource with *image* and *request*.

> NOTE
> This algorithm is intended to be called when a user agent prepares to present an installation prompt or creates an application context. User agents do not typically call this algorithm during regular page loads. Calling this algorithm requires that the client be set (i.e., the manifest was processed in the context of a document), which ensures that Content Security Policy and service worker fetch interception remain in effect. A normative algorithm for when to call this is tracked in issue #1216.

§ 2.1 `purpose` member

The ***`purpose`*** member is an unordered set of unique space-separated tokens. The allowed values are the icon purposes.

When a manifest image resource is used as an ***icon***, a developer can hint that the image is intended to serve some special purpose in the context of the host OS (i.e., for better integration). User agents *SHOULD NOT* use an icon other than for its stated icon purpose.

> NOTE
>
> For example, an icon with purpose "monochrome" could be used as a badge or pinned icon with a solid fill, visually distinct from an application's full color launch icon. The user agent uses the value of the `purpose` member as a hint to determine where and how an `purpose` is displayed. Unless declared otherwise by the developer, a user agent can use an icon for any purpose.

The ***icon purposes*** are as follows:

monochrome
: A user agent can present this icon where a monochrome icon with a solid fill is needed. The color information in the icon is discarded and only the alpha data is used. The icon can then be used by the user agent like a mask over any solid fill.

maskable
: The image is designed with icon masks and safe zone in mind, such that any part of the image that is outside the safe zone can safely be ignored and masked away by the user agent.

any **(default)**
: The user agent is free to display the icon where no `purpose` is required. For example, a manifest image resource with a "any" purpose wouldn't be used in a context where "monochrome" is required.

The ***icon purposes list*** is the list « "monochrome", "maskable", "any" ».

> NOTE
>
> If an icon contains multiple purposes, it could be used for any of those purposes. If none of the stated purposes are recognized, the icon is totally ignored. For example, if an icon has purpose `"monochrome fizzbuzz"`, then it could be used as a monochrome icon, as `"monochrome"` is a valid purpose. However, if an icon just has the purpose `"fizzbuzz"`, then it will be ignored.

To ***determine the purpose of an image***, given ordered map *json*:

1. If *json*["purpose"] doesn't exist or if *json*["purpose"] is not a string:
    1. Return set « "any" ».
2. Let *keywords* be the result of split on ASCII whitespace *json*["purpose"].
3. Let *purposes* be new set.
4. For each *keyword* of *keywords*:
    1. If icon purposes list doesn't contain *keyword*, then continue.
    2. Otherwise, append *keyword* to *purposes*.
5. If *purposes* is empty, then return failure.
6. Return *purposes*.

§ 2.2 Content security policy

The security policy that governs whether a user agent can fetch an icon image is governed by the `img-src` directive [*CSP3*] associated with the manifest's owner `Document`.

> EXAMPLE 12: Content security policy of icons
>
> For example, given the following `img-src` directive in the `Content-Security-Policy` HTTP header of the manifest's owner `Document`:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Security-Policy: img-src icons.example.com

<!doctype>
<html>
<link rel="manifest" href="manifest.webmanifest">
```

> And given the following `manifest.webmanifest`:

```
{
  "name": "custom manifest",
  "start_url": "https://boo",
  "icons": [
    {
      "src": "//icons.example.com/lowres"
    },
    {
      "src": "//other.com/hi-res"
    }
  ]
}
```

> The fetching of icon resources from `icons.example.com/lowres` would succeed, while fetching from `other.com/hi-res` would fail.

§ 2.3 Icon masks and safe zone

Some platforms have their own preferred icon shape, but as web applications should work across multiple platforms, it is possible to indicate that an icon can have a user-agent-specified mask applied by adding the maskable purpose. This allows the platform to ensure that the icon looks well integrated with the platform, and even apply different masks and background colors in different places throughout the platform.

The ***safe zone*** is the area within a maskable icon which is guaranteed to always be visible, regardless of user agent preferences. It is defined as a circle with center point in the center of the icon and with a radius of 2/5 (40%) of the icon size, which means the smaller of the icon width and height, in case the icon is not square.

> NOTE
>
> Designers of maskable icons will want to make sure that all important parts are within the safe zone.

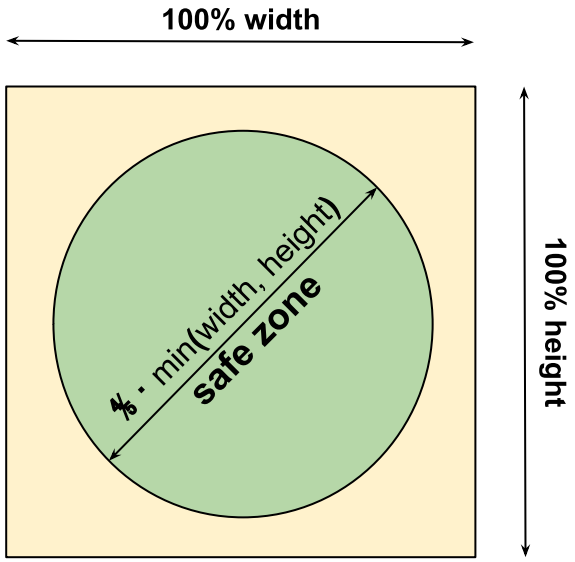

Figure 1 The safe zone is a centrally positioned circle, with radius 2/5 (40%) of the minimum of the icon's width and height.

All pixels in this zone are guaranteed to be seen in all masks. Pixels outside the safe zone are not guaranteed to (but can) be visible depending on the applied mask.

The user agent *MAY* apply a mask of any size, making any pixels that are more than 2/5ths of the image size (minimum of width and height if non-square) away from the center (the safe zone) transparent.

The user agent *MUST NOT* make any pixel within the safe zone transparent.

The user agent *MAY* enlarge the icon by adding additional padding.

If the icon contains transparent pixels, the user agent *MUST* composite the icon onto a solid fill (e.g., white) of the user agent's choice.

> NOTE
>
> It is suggested that designers avoid using transparent pixels in maskable icons.

§ 2.3.1 Examples of masks

> NOTE
>
> By staying inside the safe zone, most icons will have around 10% padding on the top, bottom, right and left with no content or non-essential content, such as an icon background. It is suggested that developers check their icon when all but the safe zone is masked out.

§ *2.3.1.1 ICONS WITH "MASKABLE" PURPOSE*

Figure 2
IMAGE
The base image with transparent background

Figure 3
SAFE ZONE
Circle with radius 2/5 (40%) of the icon size

§ *2.3.1.2 MASK EXAMPLES*

Figure 4
ROUNDED SQUARE
Android

Figure 5
SQUIRCLE
Android

Figure 6
CIRCLE
Android

Figure 7
ROUNDED SQUARE
iOS

Figure 8
FULLBLEED
Windows

§ 2.4 Monochrome icons and solid fills

Some platforms enforce that icons be displayed with a ***solid fill*** such as a single color, where only the transparency of the icon can be declared in a manifest. As web applications need to work across multiple platforms, it is possible to indicate that an icon can have a user-agent-specified color applied by adding the monochrome purpose. This allows the platform to ensure that the icon looks well integrated with the platform, and even apply different colors and padding in different places throughout the platform.

When presenting a monochrome icon, the user agent *MUST NOT* independently display the red component, green component, or blue component of a pixel. The user agent *SHOULD* display each pixel with its original alpha value, but with a red, green, and blue value of the user agent's choosing. It is *RECOMMENDED* that the user agent use the same color value for all pixels.

> NOTE
>
> Designers of monochrome icons could set all pixels to black and only use transparency to create a silhouette of their icon.

The user agent *MAY* enlarge the icon by adding additional padding.

The user agent *MAY* add a background of any color behind transparent pixels, and *SHOULD* ensure that the background has sufficient contrast with the icon.

§ 2.4.1 Example usage of monochrome icons

§ *2.4.1.1 USAGE EXAMPLES*

Figure 9
IMAGE
The base image with no color.

Figure 10
GRADIENT FILL
The image filled in with a gradient.

Figure 11
SOLID COLOR FILL WITH PADDING
Filled in with the theme color from the manifest.

§ 2.5 Processing image resources

To ***process image resources***, given list *images*, ordered map *map*, *manifest URL* and string *member*:

1. Let *imageResources* be a new list.
2. Set *map*[member] to *imageResources*.
3. If *images* is not list, return.
4. For each *potential image* of *images*:
    1. Let *image* be the result of running process an image resource from JSON given *potential image* and *manifest URL*.
    2. If *image* is failure, continue.
    3. Let *purposes* be determine the purpose of an image passing *potential image*.
    4. If *purposes* is failure, continue.
    5. Set *image*["purpose"] to *purposes*.
    6. Append *image* to *imageResources*.

§ 3. Shortcut items

Each ***shortcut item*** is an ordered map that represents a link to a key task or page within a web app. It has the following members:

- `name`
- `short_name`
- `description`
- `url`
- `icons`

A user agent can use these members to assemble a context menu to be displayed by the operating system when a user engages with the web app's icon. When the user invokes a shortcut from the operating system menu, the user agent *SHOULD* run launching a shortcut.

§ 3.1 `name` member

The shortcut item's ***`name`*** member is a string that represents the name of the shortcut as it is usually displayed to the user in a context menu.

The `name` member is a localizable member.

§ 3.2 `short_name` member

The shortcut item's ***`short_name`*** member is a string that represents a short version of the name of the shortcut. It is intended to be used where there is insufficient space to display the full name of the shortcut.

The `short_name` member is a localizable member.

§ 3.3 `description` member

The shortcut item's ***`description`*** member is a string that allows the developer to describe the purpose of the shortcut. User agents *MAY* expose this information to assistive technology.

The `description` member is a localizable member.

§ 3.4 `url` member

The shortcut item's ***`url`*** member is a URL within scope of a processed manifest that opens when the associated shortcut is activated.

§ 3.5 `icons` member

The shortcut item's ***`icons`*** member lists images that serve as iconic representations of the shortcut in various contexts.

The `icons` member is a localizable member.

§ 3.6 *Launching a shortcut*

When shortcut item *shortcut* having *manifest* is invoked, run the steps to launch a web application with *manifest* and *shortcut.url*.

§ 3.7 Processing shortcut items

To ***process a shortcut***, given ordered map *item*, URL *manifest URL*, URL *scope*, and text-direction *defaultDirection*:

1. Return failure if it's the case that:
    - *item* is not ordered map.

- *item*["name"] doesn't exist.
- *item*["name"] is the empty string.
- *item*["url"] doesn't exist.
- *item*["url"] is not a string.

2. Let *url* be the result of parsing *item*["url"] with *manifest URL* as the base URL.
3. If *url* is failure, return failure.
4. If *url* is not within scope of *scope*, return failure.
5. Let *shortcut* be *ordered map* «["url" → *url*, "name" → *item*["name"]]».
6. Process a `*_localized` text member passing *item*, *shortcut*, "name_localized", and *defaultDirection*.
7. If "short_name" exists in *item*, and *item*["short_name"] is a string, set *shortcut*["short_name"] to *item*["short_name"].
8. Process a `*_localized` text member passing *item*, *shortcut*, "short_name_localized", and *defaultDirection*.
9. If "description" exists in *item*, and *item*["description"] is a string, set *shortcut*["description"] to *item*["description"].
10. Process a `*_localized` text member passing *item*, *shortcut*, "description_localized", and *defaultDirection*.
11. Process image resources passing *item*["icons"], *shortcut*, *manifest URL*, and "icons".
12. Process a `*_localized` image resource member passing *item*, *shortcut*, "icons_localized", and *manifest URL*.
13. Return *shortcut*.

§ 4. Installable web applications

Any website is an **installable web application**.

A user agent can provide a way for the end-user to ***install*** a web application on the end-user's device, allowing the user to instantiate a new top-level traversable with the manifest's members applied.

Once a web application is installed it is known as a ***installed web application***: That is, the manifest's members, or their defaults, are applied to the top-level traversable of the web application. This

distinguishes an installed web application from a traditional bookmark, as opening a web page from a traditional bookmark will not have the manifest's properties applied to it.

> NOTE
>
> For example, on user agents that support installation, a web application could be presented and launched in a way that, to the end-user, is indistinguishable from native applications: such as appearing as a labeled icon on the home screen, launcher, or start menu. When launching a web application, the manifest is applied by the user agent to the top-level traversable prior to the start URL being loaded. This gives the user agent an opportunity to apply the relevant values of the manifest, possibly changing the display mode and screen orientation of the web application. Alternatively, and again as an example, the user agent could install the web application into a list of bookmarks within the user agent itself.

§ 4.1 Application's name

The ***application's name*** is derived from either the `name` member or `short_name` member. The user agent *SHOULD* first resolve the localized values from their corresponding `*_localized` members.

When either the `name` member or the `short_name` member is missing, empty, or the wrong type, a user agent *MAY* use the `name` member as a fallback for the `short_name` member or `short_name` member as the fallback for the `name` member.

If the `name` and `short_name` members are missing, empty, or the wrong type, a user agent *MAY* fallback to the `Document` to find suitable replacements for missing manifest members (e.g., using `application-name` in place of `name` or `short_name`). Alternatively, the user agent *SHOULD* assign a default name (e.g., "Untitled") that follows platform conventions. Alternatively, a user agent *MAY* allow the end-user to input some text that can serve as the application's name.

When both the `name` and `short_name` members are present, it is left up to implementations to decide which member is best suited for the space available (e.g., the `short_name` member might be better suited for the space available underneath an icon).

§ 4.2 Launching a web application

At the discretion of the operating system or user agent, run the steps to launch a web application with a processed manifest.

> NOTE
>
> This would typically take place when the user selects an installed web app from an app launching UI surface e.g., a home screen, launcher or start menu.

The steps to ***launch a web application*** is given by the following algorithm. The algorithm takes a processed manifest *manifest*, an optional URL *target URL*, an optional POST resource *POST resource* and returns an application context.

target URL, if given, *MUST* be within scope of *manifest*.

Other specifications *MAY* replace this algorithm's steps with their own steps. This replacement will take effect for all invocations of launch a web application.

> NOTE
>
> This algorithm is replaceable to allow an experimental launch_handler manifest field to configure the behavior of all web application launches. The replacement algorithm invokes create a new application context by default but under certain conditions behaves differently.

1. Return the result of running the steps to create a new application context passing *manifest*, *target URL* and *POST resource*.

The steps to ***create a new application context*** is given by the following algorithm. The algorithm takes a processed manifest *manifest*, an optional URL *target URL*, an optional POST resource *POST resource* and returns an application context.

1. If *target URL* was not given, set *target URL* to start URL.
2. Let *traversable* be the result of running the steps to create a fresh top-level traversable with *target URL* and *POST resource*.
3. Apply *manifest* to *traversable*.
4. Return *traversable*.

§ 4.3 Privacy and security considerations

It is *RECOMMENDED* that UI that affords the end user the ability to install a web application also allows inspecting the icon, name, start URL, origin, etc. pertaining to a web application. This is to give an end-user an opportunity to make a conscious decision to approve, and possibly modify, the information pertaining to the web application before installing it. This also gives the end-user an opportunity to discern if the web application is spoofing another web application, by, for example, using an unexpected icon or name.

It is *RECOMMENDED* that user agents prevent other applications from determining which applications are installed on the system (e.g., via a timing attack on the user agent's cache). This could be done by, for example, invalidating from the user agent's cache the resources linked to from the manifest (for example, icons) after a web application is installed - or by using an entirely different cache from that used for regular web browsing.

§ 4.4 Uninstallation

User agents *SHOULD* provide a mechanism for the user to remove an installed web application application.

It is *RECOMMENDED* that at the time of removal, the user agent also present the user with an opportunity to revoke other persistent data and settings associated with the application, such as permissions and persistent storage.

§ 5. Navigation scope

The ***navigation scope of a manifest*** is the "`scope`" member of a processed manifest. The navigation scope is the URLs to which an application context can be navigated while the manifest is applied.

> NOTE
>
> If the `scope` member is not present in the manifest, it defaults to the parent path of the `start_url` member. For example, if `start_url` is `/pages/welcome.html`, and `scope` is missing, the navigation scope will be `/pages/` on the same origin. If `start_url` is `/pages/` (the trailing slash is important!), the navigation scope will be `/pages/`.
>
> Developers should take care, if they rely on the default behavior, that all of the application's page URLs begin with the parent path of the start URL. To be safe, explicitly specify `scope`.

A URL *target* is said to be ***within scope*** of URL *scope* if the following algorithm returns `true`:

1. If *target* and *scope* are not same origin, return `false`.
2. Let *scopePath* be the concatenation of *scopes*'s path using / U+002F SOLIDUS as the separator.
3. Let *targetPath* be the concatenation of *target*'s path using / U+002F SOLIDUS as the separator.
4. Return a boolean indicating whether *targetPath* starts with *scopePath*.

A URL *target* is ***within scope*** of a *manifest* if the *target* is within scope of *manifest*'s navigation scope (i.e., within scope of *manifest*'s `scope` member).

> NOTE: Scope is a simple string match
>
> The URL string matching in this algorithm is prefix-based rather than path-structural (e.g. a target URL string `/prefix-of/resource.html` will match an app with scope `/prefix`, even though the path segment name is not an exact match). This is intentional for consistency with Service Workers. To avoid unexpected behavior, use a scope ending in a `/`.

If the application context's active document's URL is not within scope of the application context's processed manifest, the user agent *SHOULD* show a prominent UI element indicating the URL or at least its origin, including whether it is served over a secure connection. This UI *SHOULD* differ from any UI used when the URL is within scope of the application context's processed manifest, in order to make it obvious that the user is navigating off scope.

NOTE

Nothing prevents an application context from navigating to a URL that is outside of the manifest's navigation scope, while still having the manifest applied to it.

Unlike previous versions of this specification, user agents are no longer required or allowed to block off-scope navigations, or open them in a new top-level traversable. This practice broke some sites that navigate to an off-scope URL (e.g., to perform third-party authentication). See Issue 646.

§ 5.1 Security considerations

This section is non-normative.

The above recommendation (to show some UI when the application context is navigated to an out-of-scope URL) is for security reasons. It ensures that users are always aware of which origin they are interacting with.

Despite this, there is still a potential spoofing risk, if an installed app pretends to navigate to an out-of-scope site on another origin. The site shows a fake version of the user agent's prominent out-of-scope UI, indicating to the user that it is on another origin. However, in reality, the user has never navigated away from the installed app's origin, and the user agent is not showing any out-of-scope UI. User agents could try to show the out-of-scope UI in a way that cannot be spoofed by the installed app. However, due to the nature of the user agent's UI being minimal or non-existent for installed apps, this may not be possible.

§ 5.2 Deep links

A ***deep link*** is a URL that is within scope of an installed web application's processed manifest.

An application context can be instantiated through a deep link, in which case, the manifest is applied and the deep link is loaded within the context of a web application.

NOTE

The concept of a deep link is useful in that it allows hyperlinking from one installed web application to another. This can be from a native application to an installed web application and vice versa. Theoretically, this can provide seamless context switching between native and web applications through standard hyperlinks. And in the case where a particular web application is not installed, the OS can just open the link in the user's preferred web browser.

Implementers are encouraged to make such context switching obvious to the user, for example, by adhering to the human interface guidelines of the underlying platform with respect to application switching.

§ 6. Display modes

A ***display mode*** represents how the web application is being presented within the context of an OS (e.g., in fullscreen, etc.). Display modes correspond to user interface (UI) metaphors and functionality in use on a given platform. The UI conventions of the display modes are purely advisory and implementers are free to interpret them how they best see fit.

This specification defines the following display modes:

fullscreen
: Opens the web application with browser UI elements hidden and takes up the entirety of the available display area.

standalone
: Opens the web application to look and feel like a standalone native application. This can include the application having a different window, its own icon in the application launcher, etc. In this mode, the user agent will exclude standard browser UI elements such as an URL bar, but can include other system UI elements such as a status bar and/or system back button.

minimal-ui
: This mode is similar to standalone, but provides the end-user with some means to access a minimal set of UI elements for controlling navigation (i.e., back, forward, reload, and perhaps some way of viewing the document's address). A user agent can include other platform specific UI elements, such as "share" and "print" buttons or whatever is customary on the platform and user agent.

***browser* (default)**

Opens the web application using the platform-specific convention for opening hyperlinks in the user agent (e.g., in a browser tab or a new window).

> NOTE
>
> The fullscreen display mode is orthogonal to, and works independently of, the *Fullscreen API Standard*. The fullscreen display mode affects the fullscreen state of the browser window, while the [*FULLSCREEN*] API operates on an element contained within the viewport. As such, a web application can have its display mode set to fullscreen, while `document.fullScreenElement` returns `null`, and `fullscreenEnabled` returns `false`.

Once a manifest is applied to a top-level traversable, the display mode in effect is the ***applied display mode*** for that top-level traversable. The user agent *MAY* change the applied display mode for security reasons (e.g., the top-level traversable is navigated out of scope) and/or the user agent *MAY* provide the user with a means of switching to another display mode.

When the `display` member is missing, or if there is no valid `display` member, the user agent uses the browser display mode as the applied display mode. As such, the user agent *MUST* support the browser display mode.

Every display mode has a ***fallback chain***, which is a list of display modes. The fallback chain for:

1. browser is «».
2. minimal-ui is « "browser" ».
3. standalone is « "minimal-ui", "browser" ».
4. fullscreen is « "standalone", "minimal-ui", "browser" ».

The ***steps for determining the web app's chosen display mode*** is given by the following algorithm. The algorithm takes a processed manifest *manifest* and returns a display mode.

1. processing extension-point: process any proprietary and/or other supported display modes at this point in the algorithm.
2. If the user agent supports *manifest*["display"], then return *manifest*["display"].
3. For each *fallback_mode* of the fallback chain of *manifest*["display"]:
    1. If the user agent supports the *fallback_mode*, then return *fallback_mode*.
4. Assert: this step is never reached.

NOTE

The above loop is guaranteed to return a value before the assertion, due to the fact that browser is in every mode's fallback chain, and the requirement that all user agents support the browser display mode.

EXAMPLE 13

SuperSecure Browser (a fictitious browser) only supports the `minimal-ui` and `browser` display modes, but a developer declares that they want `fullscreen` in the manifest by using the `display` property. In this case, the user agent will first check if it supports `fullscreen` (it doesn't), so it falls back to `standalone` (which it also doesn't support), and ultimately falls back to `minimal-ui`.

The ***display modes list*** is the list « "fullscreen", "standalone", "minimal-ui", "browser" ».

A user agent *MUST* reflect the applied display mode of the web application in the `display-mode` media feature [*MEDIAQUERIES-5*].

NOTE

A user agent will expose the applied display mode — not necessarily the one declared in the manifest — via the `display-mode` media feature, accessible through CSS or JavaScript. Note that this media feature will also reflect other display modes for a web page when a manifest is not being applied. For example, if the end-user puts the page into fullscreen, then the user agent would reflect this change to CSS and scripts via the `display-mode` media feature.

§ 7. Privacy and security considerations

§ 7.1 Privacy considerations

This specification does not directly deal with high-value data. However, installed web applications and their data could be seen as "high value" (particularly from a privacy perspective).

As web applications can contain content that is able to simultaneously interact with the local device and a remote host, implementors need to consider the privacy implications resulting from exposing private information to a remote host. Mitigation and in-depth defensive measures are an implementation responsibility and not prescribed by this specification. However, in designing these measures, implementors are advised to enable user awareness of information sharing, and to provide easy access to interfaces that enable revocation of permissions.

It is *RECOMMENDED* that user agents prevent other applications from determining which applications are installed on the system (e.g., via a timing attack on the user agent's cache). This could be done by, for example, invalidating from the user agent's cache the resources linked to from the manifest (for example, icons) after a web application is installed - or by using an entirely different cache from that used for regular web browsing.

It's conceivable that a shortcut `url` could be crafted to indicate that the application was launched from outside the browser (e.g., `"url": "/task/?from=homescreen"`). It is also conceivable that developers could encode strings into the `url` that uniquely identify the user (e.g., a server assigned UUID). This is fingerprinting/privacy sensitive information that the user might not be aware of, and, like an identifier encoded into the start URL, it is not cleared when the user clears site data.

As with the start URL, it is *RECOMMENDED* that a user agent allows the user to inspect and, if necessary, modify the `url` of a shortcut, upon installation, or any time thereafter.

§ 7.2 Security considerations

As the manifest format is JSON and will be encoded using [*UNICODE*], the security considerations described in [*JSON*] and [*UNICODE-SECURITY*] apply. In addition, because there is no way to prevent developers from including custom/unrestrained data in a manifest, implementors need to impose their own implementation-specific limits on the values of otherwise unconstrained member types, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

Web applications will generally contain ECMAScript, HTML, CSS files, and other media, which are executed in a sand-boxed environment. As such, implementors need to be aware of the security implications for the types they support. Specifically, implementors need to consider the security implications outlined in at least the following specifications: [*CSS-MIME*], [*ECMAScript-MIME*], [*HTML*].

As this specification allows for the declaration of URLs within certain members of a manifest, implementors need to consider the security considerations discussed in the [*URL*] specification. Implementations intending to display IRIs and IDNA addresses found in the manifest are strongly encouraged to follow the security advice given in [*UNICODE-SECURITY*].

Developers need to be aware of the security considerations discussed throughout the [*CSP3*] specification, particularly in relation to making `data:` a valid source for the purpose of “inlining” a manifest. Doing so can enable XSS attacks by allowing a manifest to be included directly in the document itself; this is best avoided completely.

It is *RECOMMENDED* that UI that affords the end user the ability to install a web application also allows inspecting the icon, name, start URL, origin, etc. pertaining to a web application. This is to give an end-user an opportunity to make a conscious decision to approve, and possibly modify, the information pertaining to the web application before installing it. This also gives the end-user an opportunity to discern if the web application is spoofing another web application, by, for example, using an unexpected icon or name.

When the web application is running, it is *RECOMMENDED* that the user agent provides the end-user a means to access common information about the web application, such as the origin, start and/or current URL, granted permissions, and associated icon. How such information is exposed to end-users is left up to implementers.

Additionally, when applying a manifest that sets the display mode to anything except "browser", it is *RECOMMENDED* that the user agent clearly indicate to the end-user that they are leaving the normal browsing context of a web browser. Ideally, launching or switching to a web application is performed in a manner that is consistent with launching or switching to other applications in the host platform. For example, a long and obvious animated transition, or speaking the text "Launching application X".

The `display` member allows an origin some measure of control over a user agent’s native UI. After taking over the full screen, it could attempt to mimic the user interface of another application. This is also facilitated by the `'display-mode'` media feature [*MEDIAQUERIES-5*], through which a script can know the display mode of a web application.

§ A. IANA considerations

The mime type ***`application/manifest+json`*** is the ***application manifest media type***. Both the mime type and the `.webmanifest` file extension are registered with the Internet Assigned Numbers Authority (IANA).

§ A.1 Media type registration

If the protocol over which the manifest is transferred supports the [*MIME-TYPES*] specification (e.g. HTTP), it is *RECOMMENDED* that the manifest be labeled with the application manifest media type.

Type name:
application

Subtype name:
manifest+json

Required parameters:
N/A

Optional parameters:
N/A

Encoding considerations:
Same as for application/json ([*RFC7159*] section 8.1)

Privacy and security considerations:
See 7. Privacy and security considerations.

Applications that use this MIME type:
Web browsers

Additional information:

Magic number(s):
N/A

File extension(s):
.webmanifest

Macintosh file type code(s):
TEXT

Person & email address to contact for further information:
The Web Applications Working Group can be contacted at public-webapps@w3.org.

Intended usage:
COMMON

Restrictions on usage:
none

Author:
W3C's Web Applications Working Group.

Change controller:
W3C.

§ A.2 Link relation type registration

Relation Name:
manifest

Description:
Links to a manifest. A manifest provides developers with a centralized place to put metadata associated with a web application.

Reference:
https://www.w3.org/TR/appmanifest/

Notes:
See link type "manifest" for details about how a web manifest is fetched processed.

§ B. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *MUST NOT*, *OPTIONAL*, *RECOMMENDED*, *SHOULD*, and *SHOULD NOT* in this document are to be interpreted as described in BCP 14 [*RFC2119*] [*RFC8174*] when, and only when, they appear in all capitals, as shown here.

There is only one class of product that can claim conformance to this specification: a user agent.

> NOTE
>
> Although this specification is primarily targeted at web browsers, it is feasible that other software could also implement this specification in a conforming manner. For instance, search engines, or crawlers, could find and process manifests to build up catalogs of sites that potentially work as installable web applications.

§ B.1 Extensibility

This section is non-normative.

This specification is designed to be extensible. Other specifications are encouraged to define new members for the manifest. However, in doing so, please follow the conventions used in this specification. In particular, use the processing extension-point to hook into the steps for processing a manifest. Also, be sure to specify the steps for processing your particular member in the manner set forth in this specification. This will help keep this part of the platform consistent.

To allow the community to easily find extensions, please add your extensions to the Extensions Registry.

When specifying a new member, don't override or monkey patch anything defined in this specification. Also, don't assume your member will be processed before or after any other member. Keep your new member, and its processing, atomic and self contained. Note also that implementations are free to ignore any member they do not recognize or support.

If editors are writing a specification and temporarily want to patch this specification to help implementations along, file a bug so the community is informed of what editors are trying to do.

§ B.1.1 Proprietary manifest members

This section is non-normative.

Although proprietary extensions are undesirable, they can't realistically be avoided. If a user agent chooses to interpret a member in the manifest JSON that is not specified in this document, it can do so, but with care.

We encourage implementors adding proprietary extensions to consider whether they could become a standard (i.e. if it would make sense for a second user agent on a different platform to make use of that member, even if no other user agent has expressed interest right now). If so, we ask authors to design the API in a vendor-neutral way, and propose it as a standard. If the new member is truly proprietary (i.e. will only ever make sense in the context of a proprietary ecosystem), use this process, and prefix it with the short name of that proprietary ecosystem to avoid name collisions.

Do not use vendor prefixes that are intended for later removal once they become standard (those tend to stick around forever). Only use prefixes that will make sense now and into the future.

We encourage implementors to add proprietary extensions to our Extensions Registry. This allows the community to track what extensions vendors and/or the web community have defined and documented. Periodically, we will consider those extensions for standardization.

The following is an example of three hypothetical proprietary extensions.

EXAMPLE 14: Proprietary extensions

```
{
  ...
  "kpl_fancy_feature": "some/url/img",
  "gmpc_awesome_thing": { ... },
  "blitzly_site_verification": "KEY_9864D0966935"
  ...
}
```

NOTE

In this example, we have deliberately chosen (made-up) names of things that could be external sites or services, *not* names of browsers or browser vendors. These are not vendor prefixes belonging to the browser that invented them; they are prefixes of proprietary services.

§ C. Application Information

This section is non-normative.

Several members of the Web Application Manifest provide additional metadata related to how the web application may be presented in the context of a digital storefront, installation dialog, or other surfaces where the web application may be marketed or distributed. In an effort to support these use cases better, the following members have been moved into *Web App Manifest - Application Information*:

- `categories`
- `description`
- `iarc_rating_id`
- `screenshots`

§ D. Relationship to HTML's `link` and `meta` elements

This section is non-normative.

An extensive discussion of why we chose to use JSON instead of HTML `meta`/`link` tags for this specification is available on GitHub and on the www-tag list. Below is a short summary of the key points raised in those discussions.

The document format defined in this specification provides a unified means of encapsulating metadata about a Web application in a way that we hope will avoid existing pitfalls with both proprietary and [*HTML*]'s `meta`/`link` tags. Those pitfalls include:

- Developers have to duplicate the icons and application name in each page of a web site, leading to significant redundancy across pages. This is compounded if that information never gets used by the user agent (e.g., the user never bookmarks the web application).
- Spreading metadata across multiple documents can cause data to fall out of sync.
- If the metadata for a web application lives in a HTML document, that significantly increases the cost to user agents (and users) of checking for updates to the metadata of a site. Since the HTML file is likely to change often, it means that a user agent will often have to download the whole HTML file in order to check if any of the relevant meta tags have changed. If this resource contains inlined resources like JavaScript, images, or stylesheets, this could be a non-trivial download.

Although it would be unrealistic to think that this specification won't bring its own set of problems, externalizing this data in the form of a manifest solves the problems described above. These problems are solved by:

- Making the manifest externally linkable: External manifest files can be cached as external resources, saving both bytes and redundancy in the markup.
- Flexible value types: unlike HTML attributes, members of the manifest can represent data using complex types, such as objects and arrays, rather than just strings. This solves the problem of the awkward and highly inconsistent formats the values of proprietary `meta` tags are currently using, especially when a tag's value contains several sub-values.

In addition, standardizing the functionality currently provided by the various `meta` tag-based solutions within the manifest solves the problem of having proprietary and standard [*HTML*] tags that all achieve the same thing. Of course, this hinges on the standard actually getting implemented by browsers and those browsers getting widely deployed to users: if this happens, the Web community

might be able to retire many of the proprietary `meta` tags plaguing the Web at the time of writing. More information about the proprietary tags can be found in the *Use Cases and Requirements for Installable Web Apps* .

Lastly, this specification does not make the standardized solutions found in [*HTML*] redundant. When members like the `name` or `icons` is missing from the manifest, user agents can search in a manifest's owner [*HTML*] document for things like icons and the application name (or a user agent might even fallback to proprietary tags/metadata, if they are present in a document).

§ E. JSON Schema

This section is non-normative.

Developers interested in validating manifest documents can find an unofficial JSON schema for the manifest format at schemastore.org. It is licensed under Apache 2.0. It is kindly maintained by Mads Kristensen. If developers find any issues with the JSON schema, please file a bug at the SchemaStore repository on GitHub.

§ F. Internationalization Considerations

This section is non-normative.

It is expected that authors will localize the content of a manifest by using one of the following options:

Localized values in the manifest:

Authors can provide localized values for the localizable members of the manifest using the corresponding `*_localized` members (e.g., `name_localized`). Individual localized entries can be a string or a localized text object. A localized text object can specify its own natural language and text-direction metadata using its `lang` and `dir` properties, overriding any defaults provided by the manifest-wide `lang` and `dir` members.

User agents *MAY* handle localized values in one of the following ways:

Pass-through localization:

The user agent passes all localized values (the entire language map) to the host operating system during installation. When the user changes their preferred language at the OS level,

the host OS can immediately present the updated localized values (e.g., the application name) without requiring the user agent to run.

On-demand/update evaluation:
The user agent evaluates the localized values at parse time based on the user's current locale. When the user changes the system language, the app properties do not update immediately. Instead, they are updated when the user next visits the application or during a background update check, allowing the user agent to inspect and potentially request user acknowledgment for security-sensitive changes (such as a name change) before updating the host OS.

Dynamically setting the language:
This can include, for instance, asking the end-user what their preferred language is and dynamically adding or replacing the manifest link relationship to the document based on that language preference (e.g., using a URL like "manifest.php?lang=fr").

Using server-side language negotiation or geotargeting:
The server that hosts the web application could attempt to predetermine the end-user's language by using geotargeting or by performing language negotiation (e.g., via the HTTP "`Accept-Language`" header [*RFC9110*], or a custom HTTP header). For more details and best practices, see the W3C Internationalization articles Accept-Language used for locale setting and HTTP headers, meta elements and language information.

Given the options above, developers need to be mindful of the end-user's privacy with respect to their preferred language: When the end-user has explicitly indicated their language preference to a web application (i.e., when not just using the user-agent default language settings), sending the end-user's preferred language in the clear over the wire is generally not OK. Doing so would reveal personal information about an end-user. As such, developers are encouraged to use [*TLS*] to reduce the chances of pervasive monitoring of their Web applications [*RFC7258*].

§ G. Use Cases and Requirements

This document attempts to address the *Use Cases and Requirements for Installable Web Apps*.

§ H. Issue summary

There are no issues listed in this specification.

§ I. Change log

This section is non-normative.

The following are some significant changes that were made since First Public Working Draft:

- Formalize Internationalization appendix as Internationalization Consi…
- Format U+002F character references using W3C I18N template (#1235)
- Recommend user agents let users inspect and modify shortcut URLs (#1221)
- Redefine application context on top-level traversable (#1219)
- Rename "default display mode" to "applied display mode" (#1215)
- Clarify how manifest image resources are fetched (#1171)
- Clarify applying manifest to an existing browsing context (#1202)
- Add retiredDate for Diego González (#1213)
- Add `color_scheme_dark` member and themeable members (colors only) (#…
- Specify when an icon might be updated (editorial), and loosen sec-sen…
- Add retiredDate for Kenneth Rohde Christiansen (#1192)
- clarify the definition of "navigation scope", "applied", and off-scop…
- Make enums case-insensitive (#1149)
- Move `prefer_related_applications` and `related_applications` to mani…
- Add members for localization (#1101)
- Trim `dir`, `lang`, `display`, `orientation`, text and color values (…
- Remove problematic icon matching algorithm (#1120)
- Processing steps for id now removes the fragment (#1122)
- Rewrite privacy considerations on fingerprinting in start_url (#1114)
- Change "A" to "An" (#1103)
- Allow manifest processing to be invoked without going through an HTML…
- Ran tidy. (#1067)
- Describe manifest update behavior (#1011)

- Link to manifest-app-info in the README (#1030)
- Address privacy issue with start_url (#1029)
- Transfer display-mode to mediaqueries-5 (#1022)
- Add id member to manifest (#988)
- Use docURL as start_url (#991)
- Add missing shortcut icon (#979)
- Remove query & fragment from scope (#961)
- Remove WebIDL + rewrite all the things
- Revert "Editorial: Move some members to App Information Note (#900)
- Moving to data-cite for the accessible name (#920)
- BREAKING CHANGE: remove `platform` from ManifestImageResource (#913)
- Adding in some accessibility-related language (#898)
- Add privacy section for shortcuts member (#896)
- BREAKING CHANGE: Replace "badge" with "monochrome" (#833)
- Update links to the WG (#871)
- Remove beforeinstallprompt and appinstalled events. (#836)
- Set the manifest request's . #829
- Rewrite processing shortcuts algorithm to be more precise (#832)
- BREAKING CHANGE: remove serviceworker member (#825)
- Make Service Worker registration work properly (don't use outdated st…
- Add shortcuts feature to the explainer and spec (#768)
- Make BeforeInstallPrompt optional (#797)
- Various fixes to ImageResource processing algorithms: (#811)
- Rewrite installation process and install prompting logic (#790)

§ J. Acknowledgements

This section is non-normative.

This document reuses text from the [*HTML*] specification, as permitted by the license of that specification.

Dave Raggett and Dominique Hazael-Massieux contributed to this specification via the HTML5Apps project.

Claudio Gomboli for icon example images.

Indiana University Bloomington security researchers have contributed to this specification by reporting potential risks related to out-of-scope navigation.

K. Index

K.1 Terms defined by this specification

§ K.2 Terms defined by reference

[*ACCNAME-1.2*] defines the following:

accessible name

[*CSP3*] defines the following:

Content Security Policy

enforced

[*CSS-COLOR-4*] defines the following:

alpha component

sRGB

[*CSS-SYNTAX-3*] defines the following:

parsing (for `CSS`)

[*DOM*] defines the following:

document

`Document` interface

URL (for `Document`)

[*ECMA-402*] defines the following:

CanonicalizeUnicodeLocaleId

IsStructurallyValidLanguageTag

[*FETCH*] defines the following:

client (for `request`)

CORS protocol

CORS Request

credentials mode (for `request`)

destination (for `request`)

fetch

Origin

request

URL (for `request`)

[*HTML*] defines the following:

- active document (for `navigable`)
- CORS settings attribute credentials mode
- create a fresh top-level traversable
- `crossorigin` attribute (for `link` element)
- environment settings object
- `href` attribute (for `link` element)
- `link` element
- `meta` element
- `name` attribute (for `meta` element)
- navigated
- origin
- POST resource
- `rel` attribute (for `link` element)
- relevant settings object
- same origin
- theme-color (for `meta/name`)
- top-level traversable
- unordered set of unique space-separated tokens

[*I18N-GLOSSARY*] defines the following:

- language negotiation

[*IMAGE-RESOURCE*] defines the following:

- fetching an image resource
- image resources
- process an image resource from JSON
- `src` (for `ImageResource`)

[*INFRA*] defines the following:

- append (for `set`)
- Append (for `list`)
- ASCII lowercase
- Assert
- boolean type
- byte sequence
- concatenation (for `string`)
- contain (for `list`)
- continue (for `iteration`)
- exist (for `map`)
- For each (for `list`)
- is empty (for `list`)
- keys (for `map`)
- list
- ordered map
- parse JSON bytes to an Infra value
- set
- Set (for `map`)
- split on ASCII whitespace
- string
- Strip leading and trailing ASCII whitespace
- user agent

[*MEDIAQUERIES-5*] defines the following:

- display-mode (for `@media`)
- prefers-color-scheme (for `@media`)

[*MIMESNIFF*] defines the following:

- computed mime type
- JSON MIME type
- MIME type

[*PERMISSIONS*] defines the following:

- express permission

[*SCREEN-ORIENTATION*] defines the following:

- default screen orientation
- `OrientationLockType` enum

[*SERVICE-WORKERS*] defines the following:

- service worker
- Service Workers

[*URL*] defines the following:

- equal (for `url`)
- exclude fragments (for `url/equals`)
- fragment (for `url`)
- origin (for `url`)
- path (for `url`)
- query (for `url`)
- URL
- URL Parser

§ L. References

§ L.1 Normative references

[accname-1.2]
Accessible Name and Description Computation 1.2. Bryan Garaventa; Melanie Sumner. W3C. 29 May 2026. W3C Working Draft. URL: https://www.w3.org/TR/accname-1.2/

[BCP47]
Tags for Identifying Languages. A. Phillips, Ed.; M. Davis, Ed. IETF. September 2009. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc5646/

[CSP3]
Content Security Policy Level 3. Mike West; Antonio Sartori. W3C. 29 July 2026. W3C Working Draft. URL: https://www.w3.org/TR/CSP3/

[css-color-4]
CSS Color Module Level 4. Tab Atkins Jr.; Chris Lilley; Lea Verou. W3C. 28 July 2026. CRD. URL: https://www.w3.org/TR/css-color-4/

[CSS-MIME]
The text/css Media Type. H. Lie; B. Bos; C. Lilley. IETF. March 1998. Informational. URL: https://www.rfc-editor.org/info/rfc2318/

[css-syntax-3]
CSS Syntax Module Level 3. Tab Atkins Jr.; Simon Sapin. W3C. 24 December 2021. CRD. URL: https://www.w3.org/TR/css-syntax-3/

[dom]

DOM Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://dom.spec.whatwg.org/

[ECMA-402]

ECMAScript Internationalization API Specification. Ecma International. URL: https://tc39.es/ecma402/

[ECMAScript-MIME]

Scripting Media Types. B. Hoehrmann. IETF. April 2006. Informational. URL: https://www.rfc-editor.org/info/rfc4329/

[fetch]

Fetch Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://fetch.spec.whatwg.org/

[HTML]

HTML Standard. Anne van Kesteren; Domenic Denicola; Dominic Farolino; Ian Hickson; Philip Jägenstedt; Simon Pieters. WHATWG. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[image-resource]

Image Resource. Aaron Gustafson; Rayan Kanso; Marcos Caceres. W3C. 4 June 2021. W3C Working Draft. URL: https://www.w3.org/TR/image-resource/

[INFRA]

Infra Standard. Anne van Kesteren; Domenic Denicola. WHATWG. Living Standard. URL: https://infra.spec.whatwg.org/

[JSON]

The JavaScript Object Notation (JSON) Data Interchange Format. T. Bray, Ed. IETF. December 2017. Internet Standard. URL: https://www.rfc-editor.org/info/rfc8259/

[MEDIAQUERIES-5]

Media Queries Level 5. Tab Atkins Jr.; Florian Rivoal; Daniel Libby; Luke Warlow. W3C. 19 February 2026. W3C Working Draft. URL: https://www.w3.org/TR/mediaqueries-5/

[MIME-TYPES]

Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed; N. Borenstein. IETF. November 1996. Draft Standard. URL: https://www.rfc-editor.org/info/rfc2046/

[permissions]

Permissions. Marcos Caceres; Mike Taylor. W3C. 6 October 2025. W3C Working Draft. URL: https://www.w3.org/TR/permissions/

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc2119/

[RFC7159]

The JavaScript Object Notation (JSON) Data Interchange Format. T. Bray, Ed. IETF. March 2014. Proposed Standard. URL: https://www.rfc-editor.org/info/rfc7159/

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc8174/

[SCREEN-ORIENTATION]

Screen Orientation. Marcos Caceres; Léonie Watson. W3C. 24 July 2026. W3C Working Draft. URL: https://www.w3.org/TR/screen-orientation/

[UAX9]

Unicode Bidirectional Algorithm. Manish Goregaokar मनीष गोरेगांवकर; Robin Leroy. Unicode Consortium. 13 August 2025. Unicode Standard Annex #9. URL: https://www.unicode.org/reports/tr9/tr9-51.html

[UNICODE]

The Unicode Standard. Unicode Consortium. URL: https://www.unicode.org/versions/latest/

[UNICODE-SECURITY]

Unicode Security Considerations. Mark Davis; Michel Suignard. Unicode Consortium. 19 September 2014. Unicode Technical Report #36. URL: https://www.unicode.org/reports/tr36/tr36-15.html

[URL]

URL Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://url.spec.whatwg.org/

[UTS55]

Unicode Source Code Handling. Robin Leroy; Mark Davis. Unicode Consortium. 29 January 2024. Unicode Technical Standard #55. URL: https://www.unicode.org/reports/tr55/tr55-5.html

§ L.2 Informative references

[FULLSCREEN]

Fullscreen API Standard. Philip Jägenstedt. WHATWG. Living Standard. URL: https://fullscreen.spec.whatwg.org/

[i18n-glossary]

Internationalization Glossary. Richard Ishida; Addison Phillips. W3C. 17 October 2024. W3C Working Group Note. URL: https://www.w3.org/TR/i18n-glossary/

[manifest-app-info]

Web App Manifest - Application Information. Aaron Gustafson. W3C. 21 August 2023. W3C Working Group Note. URL: https://www.w3.org/TR/manifest-app-info/

[mimesniff]

MIME Sniffing Standard. Gordon P. Hemsley. WHATWG. Living Standard. URL: https://mimesniff.spec.whatwg.org/

[RFC7258]

Pervasive Monitoring Is an Attack. S. Farrell; H. Tschofenig. IETF. May 2014. Best Current Practice. URL: https://www.rfc-editor.org/info/rfc7258/

[RFC8246]

HTTP Immutable Responses. P. McManus. IETF. September 2017. Proposed Standard. URL: https://httpwg.org/specs/rfc8246.html

[RFC9110]

HTTP Semantics. R. Fielding, Ed.; M. Nottingham, Ed.; J. Reschke, Ed. IETF. June 2022. Internet Standard. URL: https://httpwg.org/specs/rfc9110.html

[SERVICE-WORKERS]

Service Workers Nightly. Monica CHINTALA; Yoshisato Yanagisawa. W3C. 23 July 2026. CRD. URL: https://www.w3.org/TR/service-workers/

[TLS]

The Transport Layer Security (TLS) Protocol Version 1.2. T. Dierks; E. Rescorla. IETF. August 2008. Proposed Standard. URL: https://www.rfc-editor.org/info/rfc5246/